DUE DATE: At START of class on Friday, July 5, 2002!

Note: I have delayed the due date by two days to Friday, because the midterm in Cmpt 275 and many of your other courses is in this time period. However, there are several disadvantages of this. One, the last assignment (debugging and final report) will have to be shortened to just 12 days duration. Two, if you cannot hand this assignment in on Friday before 4pm you will loose 15%, not just 5%, because Monday is 3 days later than Friday (i.e. 5% penalty PER day).

Note: there is some very interesting and important design hints in the three addendums at the end of this assignment. The first two are very important.

Note: Before editing your Assignment #1 or #2 to reflect any changes in the design during this assignment, be sure to 'accept' all the change tracking annotations from your previous assignment #2 changes. This deletes the change tracking so that further changes will only show the changes since you handed Assignment #2 in.

In this assignment, you will do the architectural design, the internal software interface design, and write a few test cases (don't forget these). I will not ask you to do a complete design specification document, as your project is not really large enough to warrant subsystem and process distribution issues. But you will have to go through a smaller design process based on use case implementation collaboration diagram traces to span the 'design gap', decide on a number of design issues, and document why you made the design choices you did by comparing to an alternate design. You will also have to compare files space for two different designs (depending on the semester project, this might be a 2NF vs. 3NF calculation, or a roll-up vs. split inheritance implementation). And, you will decide on a team coding and commenting convention, then write and compile the main module (which may or may not be particularly long), and write the C++ header files (or Java classes without function bodies) for the other modules that you feel will be needed. You will NOT write the function bodies except for the main function, and you will NOT link or run your program. You just must get a clean compile of the main module and implicitly any .h files it includes, and for the other modules just supply uncompiled interface .h files. In the case of Java, you MUST compile the other bodiless class files as that will catch any obvious interface syntax errors.

Warning: Student Assignment 3's prior to 95-2 that may be provided as examples in the SFU Library Reserves did not use the trace union technique, so be careful not to copy their design derivations and sections topics/formats directly.

Whenever you are writing, remember to ask yourself who your target audience is. In the case of a design specification, it is neither yourself, nor your 275 instructor. The fact that you are a designer on a big project that needs an architectural design phase means that it is unlikely you will do the low level design, coding, or testing yourself. Instead, assume you will have a team of coders and testers on your project staff to do the low level design and implementation. So you are old, gray-haired, and wise to the ways of good design and software engineering of big systems. And you are to use that wisdom to do a good maintainable (not brittle) design, and to communicate it definitively to your staff via your design document and some well documented by function-bodiless code. The target audience: low-level designers, coders, design review/inspection/test personnel, maintenance personnel (who may look at the architectural spec to orient themselves to where things are in the system), and your boss (the director of development in your company). You may assume all readers have read the requirements spec and user manual! By the way, I have met project managers who rejected high level design specs, not because the design was incorrect or sub-optimal, but because it was poorly documented and thus hard to review for correctness and sensibility!

Also, assume you are going to do the design, and you have been told you are then going to be transferred to England. Your document should communicate your design definitively enough so that the low-level designers and coders will not have to keep calling you in England. It will also be accompanied by enough explanatory text and comments justifying your design (especially as compared to other alternative designs) so that they will not be tempted to change your perfect design. You want to prevent them from changing, or wasting time thinking about changing the design to a different one that you have already ruled out for good reason. The reasons may not be apparent to them at first glance, unless you point these reasons out! The end of the architectural portion of this assignment results in the documentation of each individual scenario implementation and why you chose your trace implementation (vs. alternatives), and an overall Object Communication Diagram. Obviously this will include added reactive software modules (possibly representing classes) other than just the classes corresponding to your database entities. Note that an implementation normally has 3 kinds of modules or classes: database 'entity' related ones, orchestrating 'control' ones, and so called 'boundary' ones. Boundary modules or classes are user interface or system I/O related (e.g. a special class of GUI dialog box, or a class/module that handles incoming network packets. These classes provide the functionality at the outer boundaries of your Level 0 DFD.

The next portion of the assignment is internal interface design. I believe the goal of internal interface design should be a complete 'definition' of how each module appears to programmers which are coding OTHER modules! This is easily documented with extremely well commented interfaces files. In C++, these are header files. In Java, these will likely be classes with empty function bodies (though java does have an 'interface' concept, it is only for instances and therefore cannot document static attributes and static methods. Once these are all available at the end of Assignment #3, then in Assignment #4 other team members can immediately begin writing their module's implementations with statements which call your module appropriately. The team call all work at coding in parallel, at the same time. If the interfaces are well designed, and don't reveal a lot about the internal implementation (e.g. list vs. tree), the resulting implementation modules will be fairly abstract, self-sufficient, and adaptable.

By exact interface of a modules, I mean the interface another programmer needs to know:

- the names and return types of all exported functions and constructors.
- the number, type, and name of all function parameters.
- the exceptions that might be thrown by each function.
- names and types of any global constants.
- name and types of global variables (global variables should be rare).
- names and definition of named global types and classes (quite common, so others can send your functions the right type of parameters).
- other things like name and format of shared files usable by other modules.

By global, I mean exported to/known to at least one other module. The interface then, contributes to the 'export space/dictionary/pool' of program elements which any other module's programmer could choose to import/include/use. This space could be very large in sloppy 'C', where most things are global unless you take the trouble to declare them to be 'static'. There will be some of Russ Tront's Cmpt 212 notes on C++ scope near the back in coursepak B listed under Tront. Even Java programmers might want to read this, however Java tends to control its scope just with class 'private', 'protected', and 'public' keywords, where as C++ additionally uses the keyword 'static' for scope as well. Note that the keyword 'static' has 5 meanings in C++ depending on context, and some of them, in particular as I am discussing here, have nothing to do with the way that Java uses the 'static' keyword. Note that Java has another kind of

scope called package scope, which is indicated by the lack of any of the 'private', 'protected', or 'public' keywords (you can see sections 7.5 and 14 of my Jcourse lectures in http://www.cs.sfu.ca/CC/101/tront/lectureNotes/ for more info on java scope). Generally, you should choose to export a minimum of entities, and you will loose marks if you do not restrict scope appropriately. Don't export functions that are not needed outside the module they are defined in, and for each non-functional item exported you should justify its need to be global by explaining in a comment in the C++ header (or Java class) file what other module might need it and why. C++ programmers in particular are warned that we will be watching to see if you in particular know how to restrict the unfortunate default global scope of functions and variables not inside functions or classes in C++. C++ programmers should make sure they read my Cmpt 212 notes on scope.

Also, your language may throw exceptions when you read to end of file and for other problems. You may even what to program your own exceptions. One of the purposes of exceptions is to be able to write a module that detects a problem (like end of file) but doesn't know what to do with it, but forces the caller to either deal with it. The calling function often doesn't know how to detect the problem but it knows what to do about it. More than any other language, Java forces you to catch or declare the throwing of exceptions. A least a couple of people on each team should understand exceptions, preferrably all of you. If you need to learn more, see http://www.cs.sfu.ca/CC/101/tront/lectureNotes/Jcourse.s12.exceptions.pdf.

The interface design also includes instructions and restrictions on how importers should use a module's exported elements (e.g. whether the initialization function should be called before all the other functions listed in the header file. Or that the global variable Done should never be changed by code in a client module). This information is included as comments within the interface definition files.

I want you to realize that by defining the types of parameters used by procedures, you are NOT defining how that parameter info is to be stored in a hidden data structure/file encapsulated by the module boundary. i.e. If a date is passed to a module as type String (say an array of 8 characters "91/02/25"), this does NOT mean that it could not be converted inside the function and stored as three separate unsigned short integers, or as a linked list of characters! Realizing this will ease your mind when you feel you are going out on a limb by defining interfaces for which the internal representations are not yet determined. Do not create interfaces that commit the implementation module coders in Assignment #4 to a particular design. Type definitions for the entities used in the interface are just that, interface types, not storage types! Internals are not your problem as a high level designer, so don't worry about it and especially don't let it distract/clutter your mind. (Also, keeping this in mind may help you as a coder later; you don't have to store things with the same representation they are passed into a function!). So don't worry too much about how the interface will affect the implementation module code. A large system may have 100s of modules and 1000s of procedures. You can't worry about the implementation at this point. Instead, worry about what are the cohesive chunks of the system (i.e. modules and classes) and about what functionality each needs to provide to the other chunks of your team's program. <u>Concern yourself with how the nature of the interface and coupling between modules will affect the organization, architecture, and cleanliness of the system design</u>.

Function calls are mainly used to send in, and/or receive data from an abstraction. To determine the list of procedures required to be exported from each module or class, use the union of all use case scenario traces to determine what services are needed by the various clients of a particular module. Then give the functions exact names and think about WHAT info should be passed, and the DIRECTION parameters need to go: in only, in/out (pass by reference), out (pass by reference), or function return value). You MUST comment in the prototypes in your header files (or in your java class files) whether parameters passed by reference or pointer are

in/out or just out (other programmers should not have to scan around the code to find this out). Finally, decide what a suitable type would be to serve as the INTERFACE representation of each parameter passed in each function call!

To start:

1) You must identify all the modules that will participate in the Object Communication Diagram (OCD). Before doing so, it is best to decide whether some entity objects in your ORD might not be present in the implementation. So, do some calculations to decide which file design uses less space. Depending on the semester's project, this may be a 2NF vs. 3NF comparison, or a roll-up vs. split comparison.

Semesters other than 1993-3: Re-do the space calculation you did for the retained data in the requirements spec, but this time use 2NF. Show in your design document which uses less, 3NF or 2NF, and the assumptions underlying your conclusion. If 2NF uses less space, and you don't think the time and trouble of entering/maintaining redundant data is a problem, you would eliminate the 3NF objects and use 2NF. In fact for Cmpt 275 use 2NF anyway, even if it uses more space, because it will make your 275 project simpler. You must show the 2NF calculation and state which uses less space, but your reason for choosing 2NF can be "the instructor told us to use 2NF".

Semester 1993-3: Calculate the space needed for 1000 cargo items, 10% of which are pallets. Do this calculation assuming roll-up, then compare it to a calculation using split. In an appropriate section of your design document, show the calculation, indicate the assumptions underlying your conclusion (do not assume the reader has read this assignment sheet), and state which uses less space. If roll-up uses less space, use it and say that is why. If split uses less space, indicate that but state you are using roll-up because "to save the programmers work, the instructor told us to use roll-up!".

2) Abstraction dictates that we try to hide the persistent (i.e. disk) storage of (vehicles, cargo items, whatever applies in your semester) of all types, each in their own single module. So, for the time being, think of both superclass and subclass items as being together in one module. In Assignment #4, you can refine how this is handled (roll-up, roll-down, or split).

Note: students often make a mistake here about what their later implementation will be like, which wrongly dictates some aspects of their architectural and interface design. They *wrongly* assume that on program start-up, they simply open and read in all the data from a file and store it in a linked list in RAM. You must consider that the data files might be large and will not fit in RAM. Also, what would happen if the power failed while the updated data was in RAM? You must learn to read and write individual binary records, in place, on disk! You must use random access, binary I/O for all your files in Cmpt 275. The instructor has notes on the class web site on how to do random access binary I/O in both C++ (advanced.io.pdf) and in Java (BIOJava.rev.11.pdf). At least two of your team members should start reading these right away. Though you won't use these techniques until assignment #4, the fact that you will be using them somewhat affects how you should do your design in Assignment #3.

Also, in some semester's projects, you may find that adding 'derived attributes' to some of the objects may allow a considerable increase in performance. Actually, you may already have put some derived attributes in your requirements spec. There may or may not (depending on the semester) be opportunities for more. For any that you could potentially use, you must discuss their advantage over not using them.

3) Your OCD will contain a main module, one module for each entity class in your ORD, and possibly UI and/or scenario control modules. To decide whether to include the latter modules, first decide on where you are going to implement your menu code (in the main, in a UI

module, or split among main and scenario modules (e.g. perhaps a sub-menu in the scenario module that implement the use case operations from that sub-menu)). Document in your design document why you chose the menu code placement you did for this project, compared to some other alternative. Then you will want to decide the following 3 design issues in conjunction with each other:

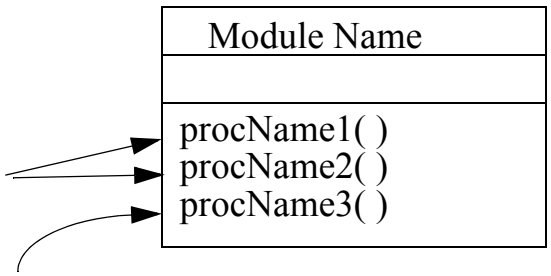
- whether you are going to use centralized scenario control, roundabout control, or principle object control.
- whether you will use special mid-level control functions/modules added specifically to contain the scenario orchestration code.
- what UI library scope you are going to allow.

Document why you made your particular choices, in comparison to other options.

4) Lay the module icons out on a blank OCD. You may leave system-provided modules (such as file I/O, etc.) out of the chart to make the diagram a little simpler. Leave room at the bottom for pseudo-code scenario descriptions. Make many copies of the resultant diagram. You are to hand in one 'single scenario' OCD/collaboration diagram for each scenario. You may hand draw the call traces on each scenario collaboration diagram, and hand write in the pseudo-code description (on the bottom of the single scenario diagram or on the next page). Make sure you label the arrows with the call precedence number. If possible, show these numbers within the pseudo-code.

Discuss somewhere in your design document which modules will be responsible for initialization/shutdown of others. Where in the hierarchy can decisions to reset or end the program originate? How will any necessary initialization and shutdown control be signalled throughout the OCD to each module needing it? It is considered bad design if one module calls (and therefore knows about) all the others; each module typically knows only about the few below it or beside it.

5) Draw an overall OCD by taking the union of all the single scenario collaboration diagrams. Remember, you will not need sequence numbers. The overall OCD will determine the services/functions required to be exported from each module. In the new drawing, show the module icons like the following:



The gap in the middle of the object represents the place that we put the attributes if we wanted to show them (we don't here, as we are concerned only with functional services in this part of the design process). You may want to draw the call arrows to the correct function name as shown above (then you don't have to label the arrows with procedure names). If this convention becomes too spaghetti-like, use the other side of the box too, or revert to arrows labelled with function names that hit the module icon anywhere on its perimeter. Optional: If you want, you can now additionally put these functions names into the bottom section of the class icons in the ORD shown in the prototype section of your requirements spec.

6) Agree on a coding standard (see more below). You can document this coding standard either in your design spec, or as comments at the top or bottom of your main program. Most

Cmpt-275 teams choose to put it at the bottom of their main module. Make sure you decide how you are going to document the revision history of each module in the comments within that module.

7) Start writing the interface (C++ .h or Java class) files. At the top of each write a sentence or two describing each module or class's general purpose/exported services, and what it contains/hides (e.g. Abstract Data Types, sub-menus, user I/O). More information on commenting is supplied further below. By the way, you should write the class instance and static attributes in your classes now, unless they are private in scope. This is because another module that instantiated (or subclassed) your class could get access to the non-private attributes, and that other module's programmer may want to start using your class soon! However, that programmer need not know about how you store entity attributes on disk. He/she only has to know the *non-private* instance and static attributes in RAM which he/she might use because you have not made them private.

8) Begin detailed interface design by writing (on paper or in the .h file) the procedure parameter lists for each exported procedure, and decide the directional nature of each parameter: in only, in/out, or out only. Define the representation of each user defined global type, global variable, and each functions formal parameters.

Also, give widely used types a type name, or constants a name.

e.g. const CUSTOMERSIZE = 30;

typedef char CustomerArray[CUSTOMERSIZE];

A programmer-defined type name like 'CustomerArray' above can be defined once in a particular header file (which one?), then exported to everywhere it is needed. Then, if this type should ever have to change (say change CustomerArray to a C++ String class), then it likely only need be changed in one header file and this will (perhaps smoothly) affect all the other modules that include it. Note that it is also good to define constants that might be used in several modules for loop limits or screen output widths as global constants. That way by changing the 30 above to 40, not only do you increase the size of the array, but also change the limit of all loops that traverse that array!

Note #1: Do not put all the global definitions in one .h file. Instead, put each one in the .h file or class to which is most related.

Note #2: Often students like to put constants highly related to a C++ class inside the C++ class itself. Then you refer to it the same way you would to an attribute variable in the shepherd's brain. This is good cohesion. But if you need these constants at compile time to dimension the size of arrays, C++ generally restricts inside constants to be integer only. See the document classConstants.pdf on the Cmpt-275 web site if you are working in C++. Note that in Java, it is possible to advertise global constants in an 'interface' file.

Example Table of Contents for the Architectural Design Document

(Don't forget the 3 release pages and other revised documents that come before this one in your assignment submission. Don't forget to turn on change bars before revising the previous documents.). And don't forget the labelled-tab dividers before each document.

i) Architecture Document Title Page

ii) Architecture Document Revision History Page

iii) Architecture Document Table of Contents Page

The following provides hints for what I would like in each section of the Architecture Design Document. You may assume the reader will have read the requirements spec and the user manual. Remember, write and include what you wish a project manager would leave you (if you were a coder) about his wise, high level design, before heads to England.

1.0) Introduction:

In a paragraph, introduce the reader to the application and mention the other existing documents that he should refer to for background info. In the second paragraph, describe the purpose of the Architectural Design Document. Mention that much of the interface design is not located in this document, but available by referring to the actual source code modules that are part of the release. (Put these source listings after yet further labelled-tab dividers in the back of the assignment binder, and make an entry for each in the project release table. Put a labelled-tab divider in front of each class or module's source files.)

2.0) Refinement of the Analysis Models:

In a big project, this section might show further decomposed DFDs, added sub-states, etc. However, for Cmpt-275, all I want you to show is the calculations for disk space for 2NF vs. 3NF (or roll-up vs. split, whichever is appropriate for your semester).

Also, show and discuss the reasons for any derived attributes you may have added recently to improve the performance of certain operations.

3.0 Architecture Presentation and Discussion

Present the overall OCD (i.e. union of all collaboration diagrams). Very briefly describe the name and purpose of each module, and possibly point out any layering, major abstractions, cohesion, and fan-in present.

Discuss the design decisions you made and how they affect the visual nature of the diagram (i.e. added or merged modules). Keep in mind Section 4.0 below. In a paragraph each, discuss:

- where you will put the menu state machine code and why. Some students even put the main menu stuff in one module and distribute the submenus to other control modules. I do not think this is better or worse.
- the nature of your overall scenario design strategy and scenario control location policy: either distributed in a roundabout manner, distributed in a principle object rooted manner, centralized in the UI or scenario modules, or centralized in principle object modules. Why did you choose this architecture?

Present the list of scenarios that the system must handle, and then attach the many, single-scenario collaboration diagrams (alternately, you could use UML sequence diagrams if you wish). Don't forget each must include a scenario description possibly in pseudo-code form. Discuss which modules need initialization/shutdown, and describe/show which other modules are responsible for triggering or passing on needed initialization/shutdown control signals.

4.0) Comparison with Alternate Design

The only way I have been able to get students to actually consider design alternatives (which is the essence of design) is to insist they provide an alternate design. Give an alternate OCD diagram which illustrates a different design from the one you are going to adopt, but which actually could have been used. Generally you will want to show the effect of a major design policy change. Don't spend a lot of time using a computer to draw this; you could just sketch it with a pencil. Also, don't give the individual scenarios, just an overall OCD.

In a paragraph or two, discuss the advantages and disadvantages of this alternate design when compared to the actual design you want to use. Finally, mention why you feel your final de-

sign's advantages outweigh the alternate design's advantages (or why your final design's disadvantages are fewer than the alternate's). This is your chance to 'blow your own horn' and tell me WHY your design is so CLEAN and beautiful architecturally (not implementation or efficiency wise, but architecture-wise).

5.0) Initial Interface Design:

Justify the use of any exported constants, types, or variables. Discuss why you couldn't avoid them, or why they contribute to a more maintainable design.

6.0) Detailed Interface Design:

State that detailed interface design is not part of this document, and refer the reader to the main and header/class modules included separately in the release.

7.0 Suggestions for Lower Level Design (Optional):

As a high level architect, you may optionally include any advice for the lower level coders here. This may include estimates of database size, relative frequency of operations on certain modules, and required access speed for specific modules. (The requirements spec only spoke of these things for the overall system. This section will have broken things down to the individual module level). The coder will know for instance that if inserts to and deletes from collections are very frequent compared to inquiries and modifications, sorted files are not very desirable. You could also give book or article references to any great algorithms for particular calculations that will be needed. For Cmpt 275, you can leave section 7.0 out, or just simply say that for the prototype, files can be unsorted and use linear search (which is actually all I expect in Assignment #5; remember the goal of Cmpt 275 is not to make a hot product, but to experience a project).

8.0 Testing:

The testing section would normally suggest test strategies to be used, or test coverage suggested. You, in CMPT 275, should change the section title to Test Cases, and include 3 detailed test cases as mentioned below.

Test Cases

If test cases have not been covered yet in the course, they will be very soon. They are in Section 7.10 if you want to assign someone to start writing these right away.

I have a hard time to get students to write black box test cases before they write the code. In industry, this is often done in parallel with low level design and coding (i.e. Assignment #4), but students always seem to code first (rather than in parallel), then write the tests just before the Assignment #4 deadline. So to prove to yourself that writing black box test cases can be done before implementation code, in Assignment #3 you are to hand in 3 black box test cases. You don't have to supply a battery of tests which will completely test the system or even fully test a module/class. I only want you to demonstrate that you can write one functional test case, one stress test case, and one performance test case. These must be complete, stating what kind of error each test tests for, the exact pre-conditions of the test (exact input files, state/mode you are presently in and/or what you typed to get to the present state/mode), exactly what you type in or do to accomplish the test, and exactly what actions/outputs you expect from the test (likely an example of an expected output screen, printout, or resultant file contents). And I mean EXACT!

Source Code

The source code is *NOT* part of the Architectural Design Document!!!!!! Each module is a separate Computer Software Configuration Item (CSCI). Each CSCI must be mentioned in its own row in the Release Table, and separately put into the back of the release binder with a labeled-tab divider. Each CSCI, including source files, must be separated from the others by a labelled tab divider. You will loose marks for neatness if you do not include these dividers between each CSCI (the only exception to this rule is that you needn't divide the separate C++ header and implementation source listings of the same module). The written documents of your assignment should be first, followed by the main module source listing, then the rest of the listings in alphabetical order. (If using tractor feed continuous paper, the included listings MUST be burst (torn) into separate pages so they can be easily read while bound, otherwise you will loose marks for neatness).

Each source code module is to have a certain commenting and coding format. I will specify some of the commenting that is required. Your group is to specify the smaller details of your coding convention in your main source file comments. This will provide the needed revision information, informative commenting, and a uniform look to all modules even though they might have been written by many different coders. Studies have shown that a common coding style in a company increases the speed at which others can read and understand your code. The things to agree on are: a) how to divide major sections of a source code module, b) the commenting of modules and functions, c) the capitalization of type/variable/procedure names, d) spacing/indenting, and e) the revision history format and order.

a) Each major section of a listing should be separated by a dividing comment line.
e.g. `/*----------*/`,
`//==========`
or `/**********/`.
See my examples below. Pick one style and all members are to stick with it. Actually, better yet, use rather bold dividers for major sections (e.g. beginning of constants and types, beginning of static functions, beginning of instance functions, and use less bold ones between each function.

b) Each module will start with a comment stating that file's name, then have a Revision History section. Agree on the revision history format, and chronology (oldest first or last?). The next section should be a paragraph-long comment describing the purpose of the class/module. What 'group' of functionality does it encapsulate. It is important to state the nature of the cohesion of the module or the nature of (not implementation of) the data or control abstraction retained by/hidden in the module. You may also mention any general rules as to how the module should be used. e.g. /*You must call CreateStack() before you can use a stack*/. In contrast, advice that is specific to only one function should go in the comments of the individual functions. Again, see my example below.

c) You need a name, spelling, and capitalization convention. I will accept any standard you propose, but the whole project must use exactly the standard. For instance, is the first letter of a type or class always upper case? Is the first letter of every embedded word in a variable name capitalized? Are names of constants completely uppercase?

d) Finally, you need to decide on an indenting, spacing, and punctuation convention. Where will "{" go: below the function, switch or loop, or on the first line? Will the body of the else clause be indented 3 or 5 spaces? Will "=" be surrounded by spaces, and will there be spaces after commas and colons? e.g. from Modula-2:

```
Type Color = (Red, Green, Blue);
```

```
vs.   Type Color=(Red,Green,Blue);
```

Document your coding convention at the end of the main module. It should only take about a half page if you are concise. Before group members set off to code, they should agree on this convention, write this half page of the main, and give/e-mail each member a copy. This will prevent some poor editor from having to go through all the code, just before the assignment due date, and fix it all up! See my example format below, and choose a similar convention to adopt in your group. Your convention must include all the 'information' my example does, though you may format it differently if you want. The only exceptions are the following compulsory things:

- In C++, each individual function prototype in header files must be **immediately** preceded with a thin horizontal dividing line. You are *not* to put a line, then comments, then the function prototype/body! This is because when scanning a document, the TAs and I and most programmers want to find the function signatures quickly, not have to wade through a indefinite number of comments lines under a dividing line to find the signature. Put any specific comments regarding a whole function *after* the signature before the first statement.
- In Java, you may put the general comment about a function above the function. This is because, unfortunately, a comment extraction program (called javadoc) which automatically writes beautiful documents for a class, requires that the comment describing a function must go above the function signature in special delimiters. You do NOT have to learn to use these special delimiters (though it is really interesting to do so and I bet that at least one group will hand in the javadoc for their classes rather than the source itself). Java programmers should put a thin dividing line comment, then the function comment, then some short end-of-comment indicator (e.g. //---) so readers can quickly find the signature below an indefinite number of comment lines.
- I am going to insist that you put each function parameter within a signature on its ***own*** separate line! Each parameter line will show the parameter's formal name, its type, and its parameter direction (either in, in/out, or out). See the example below. If necessary, each parameter's line should also include the unit of measure for each parameter (e.g. radians or degrees, Celsius or Fahrenheit) if this is relevant. Don't bother to state the unit of measure for a storage module whose parameters simply take data in for later retrieval and output (Why?).

The comment for each function signature will describe what the function does, along with any tricky use, calling, or exception details for that function. Do not comment about how the function works internally, only what abstract service it provides to other programmers that might call it. By the way, in semester 93-2, a student sent me the following e-mail message:

"How much do we have to comment? For instance, I have a procedure called InitMenuBar. I can't see how much clearer I can describe it. Does it need a descriptive comment?" My answer was: "Dear Student: Does your procedure just initialize it, or also create/allocate it (possibly returning the exception 'out of memory'), and/or actually display it? Russ."

This illustrates how hard it is to extract yourself from what you know about your own design, in order to comment for a maintenance or client module coder who doesn't know what is in your mind! The best policy is to pretend you are going to give your code to your best friend, then leave the country. Assume he is new on the project and has only skimmed the requirements spec and user manual. How would you comment for him/her? Anyway, here's an example of

kind of coding and commenting style I want. It is in Modula-2, but I am sure you will get the idea. Note that the comment delimiters in Modula-2 are (* and *).

```
DEFINITION MODULE ExampleSort;

(* Revision History:
Rev. 2 - 90/02/11 Modified by Joe Smith
       - to add third parameter to procedure Sort.
         (Note: It is also good to document WHY a
         particular change was made).
Rev. 1 - 89/03/19 Original by R. Tront
(*-------------------------------------------------------------
This module hides the implementation of the sorting algorithms. The exported
procedure is called qualified by a flag as to whether the data to be sorted is
mostly sorted already.  The algorithm used for unsorted data may in future need
to change depending on the database size.  Cmpt 275: You should also mention in
the module header comment, in addition to what is being hidden/abstracted, why
these procedures and other attributes are together.  i.e. the nature of the
cohesion.*)
(*-------------------------------------------------------------*)
(*IMPORT or #include statements go here *)

(*-------------------------------------------------------------*)
(*Exported constants/types/variables*)
TYPE  Sorted = (Almost, NotVery);  (* self-explanatory *)
  WasAlready = (OK,     OrNot);    (* indicates whether array
                                      was already sorted *)

(*-------------------------------------------------------------*)
PROCEDURE Sort (
       VAR    data      : ARRAY OF INTEGER;   (* in/out *)
              sortedness: Sorted;             (* in *)
                               (* -  this parameter is a hint
                                     to the sort routine *)
       VAR    already   : WasAlready           (* out *)
                               (* -  you may document units here too if
                                     relevant (e.g. inches or meters) *)
                  );
(* Sort is a useful procedure which will sort an integer array, and given
information via the second parameter, will use a different strategy depending on
whether the data is almost sorted already, or not.  Variable 'already' returns
whether the input array was already sorted.  i.e. didn't need sorting*)

(*-------------------------------------------------------------*)
PROCEDURE OtherProc()
(* The parameters of a procedure should be listed one per line as shown above,
with a comment giving info on whether it is an input parameter, an input and
output parameter, or solely an output parameter. You can't tell this simply by
whether they are VAR/ref/pointer parameters or not! After each procedure
definition, there should be at least a 1 paragraph description of the procedure:
what it does, how to use it, what its restrictions are, etc.  You do not have to
suggest what internal algorithm a procedure will use.  This interface file is an
advertisement of only how another programmer should use an exported procedure.
Note that you do not, during interface design, have to document procedures that
are not exported! *)
END ExampleSort.
```

Assign good descriptive names to all exported program entities, modules, files. And if possible, use C++ reference types rather than pointers for in/out and out parameters.

Compilation

Modern development environments don't make it very easy to get a listing that proves your main and source files compile. If you find a way (sometimes asking for a compilation listing with line numbers and no error messages showing), please tell me. So in order to prove your code compiles (do NOT try to build or make), you must supply a floppy disk of all your code. Attach it firmly to the inside cover of your assignment, or buy a special plastic 3 ring holder to hold it in your 3 ring binder. Or scotch tape an envelope onto the inside cover of the binder, put the floppy in the envelope, and lightly seal it with a bit of scotch tape. Note that some languages do not have the concept of an interface/header file (e.g. Java, and possibly Visual Basic). But you can write the classes or modules and not give any bodies to the functions yet. And Java has a neat tool to extract the prototypes and 'special' comments from an implementation module.

NOTES:

In your spare time, though you are not supposed to think about implementation at this point, one team member could start some experiments trying to call the operating system, trying to open and use binary files with fixed length records (try overwriting a record in the middle of the file), or whatever else you are going to require support for. For C++ programmers there is information on random-access, binary I/O available in file 'advanced.io.pdf on the class web site. These preliminary implementation tests are a good task to assign to a systems programmer.

If using Unix and NT, it is not easy to have a program print directly to the printer. For Cmpt 275, I will allow you to instead just write to a file, then tell the user in the manual to use a DOS or MS-Windows command to get the file printed.

Marking

You will be marked out of 50 using the following distribution:

5 marks-	Release pages and consistency with other CSCIs, and Architectural Design Document title, revision history, and table of contents pages.
10 marks-	Excellence of design, description of design, justification of design choices.
7 marks-	Correct use of diagrams and scenario descriptions/pseudo-code.
5 marks-	Main and source file history and general/top module comments.
3 marks-	Function prototype comments (parameter and general/top comments).
10 marks-	Test cases, each with 4 parts: what being tested, exact pre-conditions, exact steps, and exact expected results (i.e. simulated screen or printer outputs).
5 marks-	Organization and neatness, choice/definition and consistent use of coding and commenting convention, order of listings, bursting of listings, labelled-tab dividers.
5 marks-	Spelling and grammar, aimed and worded at correct audience.

Addendum on File Iterators

When doing your design, many students in the past have had a hard time knowing how to structure a program that needs to get every record from a file. Perhaps this is needed for a displaying all records so the user can select one, or perhaps it is for getting them out of the file so that you can select a subset of them (e.g. all cargo items scheduled for a particular flight), or perhaps it is for a printout where you want to compose and print one line at a time. In any case, the proper concept to use is that of an iterator. Iterators can be fancy objects in themselves, however you probably just need a couple of plain C++ functions (perhaps not even in a class), or a couple of java static functions in each class that fronts for a database file.

The two functions typically needed are:

- moveToBeginningOfFile( )

- getRecord( )

The first does what its name implies.

The second reads the record currently pointed to by your file module and returns the data in an object instance. The read implicitly has the effect of moving the current position in the file ahead by one record. These are not direct random access file library functions, they are meant to be functions your module/class exports to other programmers on your team so they can access your module's/class's records. You will write these functions by calling the underlying random access binary file i/o library functions provided by your programming language.

You might also want to provide a few other functions like AddRecord( ) that takes an class instance (say ChangeRequest) as a parameter and throws it onto the end of the database. You may also need a function that deletes the next record from your current position.

By providing these four functions for each class/module that is controlling a particular database file, you will allow the scenario control functions to properly orchestrate a user command.

Note that some teams occasionally try to create one file management class, and have multiple instances of it, one for each database file. That is not trivial for 2nd year students, so I advise you do it at your own peril.

You may want to have one of your expert system programmers write one java class that fronts for a database file. She can show it to others, and then each of them can write a similar module/class for their database file. Thus each student gets the experience of writing random access binary I/O functions, but the team generally has to only waste time struggling with writing the initial one.

The reason we have a module or class front for a file is that the file format and organization may change in the future, and we do not want this to affect the rest of the program. E.g. instead of having unsorted records in the file, we could have sorted ones, or a B+ tree for faster access. The rest of the program should not be affected by such implementation changes if they are hidden from other modules/programmers by a nice module or class interface that provide adequate general functionality as described above.

Classes Fronting for Persistent Records

Students in 2nd and 3rd year often have trouble with the concept of file records. First, they are tempted to read them all into a linked list in RAM (like Cmpt 201). This is usually incorrect for a large database-like file. Second, they don't appreciate that there is no use converting a four byte 2's complement integer to a variable length (up to 10 byte) ASCII or UNICODE string for storage in a file, then changing it back to 2's complement when it is read back in (what a silly waste of conversion effort and space!). Third, they usually do not have experience doing random-access file I/O where they can move to any location in the file without reading all the bytes between their current location and where they want to get to. When they get there, unless they have read further about random-access I/O, they don't realize it is possible to overwrite one record on disk without disturbing the record before and after it. And finally, they have a hard time merging object-oriented concepts (which are only RAM related) and file records. Usually a programmer writes a class to have some kind of functions written to write a RAM instance to a file record, and also provides a function to read from disk back into a RAM instance. It is not student's fault that they do not understand these concepts.

Let's talk about the last topic above: a class that provides instances that are also stored on disk. What must such a class provide?

- Functions to create, open, and move around the file. And constructors for RAM instances.
- A function (static or instance?) to write an instance to a file (Q: to what location in the file?).
- A function to fill an instance (overwrite its RAM attributes) from a record at the current position in the file. Et cetera.

There are many strategies to read and write the file. First, you can write in binary fixed length records, or use Java's Serialization functions (which produce variable length records that have a number of drawbacks). Note in C++, writing structs and non-polymorphic class instances to disk in binary is quite easy, just give the write function a pointer to the first byte of the instance and the number of bytes to read. (However, if a C++ class has virtual functions or any of its ancestor classes have virtual functions, then there are sneaky hidden points in the instances that point to virtual function tables. If you write these to disk, and read them back another day, the virtual function table will not likely be at the previous location.)

In Java writing in binary is much harder that simply writing structs in C/C++ (see BIOJava.rev.11.pdf advice written by a previous Cmpt 275 TA). However, Java does have some nice libraries for accessing different types of formal databases. In Cmpt-275, please use the techniques shown in BIOJava.rev.11.pdf.

Second, are you going to use static functions or not? e.g. To write an instance to file, do you call an instance function and tell an instance to save itself? Or do you call a static function and pass it a reference to the instance needing saving?

To get the record for a customer named "Russ Tront" do you construct a RAM instance and set the customer name attribute to "Russ Tront" and then tell the instance to find its data on disk? The instance function then goes and reads the file until it finds the one with the correct name, and then reads the other attributes of that record into the other instance attributes of itself.

Or do you call a static function and pass it the string "Russ Tront". It constructs, fills, and returns an instance. Question: In C++, do you want this returned by value or by reference?

Or to you create an instance, set the name attribute to "Russ Tront", then pass the instance to the static function by reference. The static function will read the file and restore the attributes of the instance from disk, then return. The calling function will now notice that the instance's attributes are all properly restored from the persistant store on disk.

In fact, I would not take marks off if a student team did not have entity class instances at all. You can read a record with a static member function that returns the attributes in multiple parameters. And you can pass all the fields of, say, a customer record each in a separate parameter to a static member function for writing to disk. What do you need instances for? I am not saying this would be best, but it is not horrible. You might find that your 275 project does not have a lot of instance-oriented nature to it (many informations systems do not). If you make yours instance-oriented, you will have to work at figuring out where you are going to create, use, pass around, and destroy instances.

In Cmpt 275 Assignment #3, you have to choose between some of these alternatives (that's what design is). There are three well known types of classes: boundary classes, scenario control classes, and persistent 'entity' classes. If you are going to write an entity class that someone else on your team needs to call, then in Assignment #3 you must decide on your class's software interface. Are you going to use static functions or not? How exactly? What are the functions going to be named. What kinds of things do they pass in and out. Should your teammates not use any RAM objects after they have been written to disk, because you have deleted them? What functions are you going to provide for moving forward and back in the file. Is your whole team going to use the similar strategies for all the entity classes in your project?

Addendum - Returning Base Type Parameters

In Java, you can return parameters by reference if they are instances. You pass an instance to a function parameter, the function can mess with the instance, and when the function returns, the caller finds the instance modified. However, it is very hard to do this with Java base types like int, float, boolean. If you want to return one of these, you can use it as the return value. If you want to return several base types via the parameter list, you must use special holder classes.

Java provides wrapper classes for base types so that you can put an integer in a linked list. Java lists assume the elements in the list are (references to) object instances, not int or float. Java provides wrapper classes like Integer and Float which are instances that can hold an int or float respectively. However, these silly classes do not allow you to ever change the contained value!

Instead, you have to import the classes IntHolder, FloatHolder, et cetera, from the package org.omg.CORBA. Here is an example:

```
import org.omg.CORBA;
int value = 6;
IntHolder myIntReference = new IntHolder (value);
y = functionThatChangesValue(myIntReference);
```

Java is a beautiful language, but not everything about Java is beautiful eh?

(The three not very beautiful things about Java that I hate are keyboard input, binary I/O of instances, and passing base types by reference as shown above. Russ Tront)